Problem 1. (10 points)

Let $G = (V, E)$ be a weighted directed graph with weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer $W$. Show how one could use Dijkstra’s algorithm to compute the shortest paths from a given vertex $s$ in $O((n + m) \log W)$-time.

Answer: First we prove that at any step of Dijkstra the difference between the maximum and the minimum value for the vertices in the priority queue (i.e., outside the cloud) is no greater than $W$ (excluding nodes with label $+\infty$). We prove the statement by induction.

Base case: at the first edge relaxation (i.e., when the source is pulled in the cloud), all the neighbors of the source will have a label no greater than $W$, and therefore the difference between the values in the queue is no greater than $W$.

Induction step: assume that when the $k$-th node is pulled in the cloud, the statement is true. Let $V_{\text{min},k}$ and $V_{\text{max},k}$ the two vertices corresponding to the minimum and the maximum at this step. At step $k + 1$, we extract $V_{\text{min},k}$ from the queue. All the nodes that get relaxed will have a label $V_{\text{min},k} + w$ where $w \in \{0, 1, \ldots, W\}$. So after the update, all the values in the queue are between $V_{\text{min},k}$ and $V_{\text{min},k} + W$. Therefore, the difference between the minimum and the maximum is still no greater than $W$. QED.

Since the weights are integers, the statement above implies that the number of distinct values for $D[]$ for the nodes outside the cloud is $W + 2$ ($W + 1$ distinct integers in the range $[0, W]$ plus $+\infty$). In other words, there are at most $W + 2$ distinct shortest path estimates. Thus, the number of distinct entries in the priority queue will be $W + 2$. Nodes with the same $D[]$ value will be stored in a linked list attached to the node of the heap corresponding to the entry $D[]$. Then, Dijkstra will run in $O((n + m) \log W)$-time.

Problem 2. (10 points)

Suppose that we are given a set of $n$ objects (initially each item in its own set) and we perform a sequence of $m$ FIND-SET, and LINK operations, where all the LINK operations occur before any of the FIND-SET operations. Show that the resulting sequence will take only $O(m)$ time (if both path compression and union by rank are used).

What happens in the same situation if only the path-compression heuristic is used?

Hint: Use the accounting method. The key observation is that since all the FIND-SETS are done after the LINKs, once a node $x$ appears on a find path, $x$ will be either the root or a child of a root at all times thereafter.

Answer: Charge MAKE-SET two dollars. One pays for the MAKE-SET itself and one is left stored on the node $x$ that is created. The credit pays for the first time $x$ appears on a find path and is turned into a child of a root.

Charge one dollar to LINK. This dollar pays for the actual linking.

Charge one dollar to FIND-SET. The dollar pays for visiting the root and its child and for the path compression of these two nodes, during the FIND-SET. All the other nodes on the find path use the stored dollar to pay for the visitation and path compression. As mentioned, after the FIND-SET, all nodes on the find path become children of the root (except for the
root itself), and so whenever they are visited during a subsequence \textsc{Find-Set}, the \textsc{Find-Set} operation itself will pay for them.

Since we charge each operation either one or two dollar, a sequence of \( m \) operation is charged at most \( 2m \) dollars, and the total time is \( O(m) \).

Observe that nothing in the above argument requires union by rank. Therefore, we get an \( O(m) \) time bound regardless of whether we use union by rank or not.

**Problem 3.** (10 points)

In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of \( \{d_1, \ldots, d_k\} \) units. They seek an algorithm that will enable them to make change of \( n \) units using the minimum number of coins.

1. The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of \( n \) units using US denominations. Prove its correctness and analyze its time complexity.

2. Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are \( \{1, 6, 10\} \).

3. Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of \( n \) units using denominations \( \{d_1, \ldots, d_k\} \). Analyze its running time.

**Answer:** Here is the greedy algorithm.

**Inputs:** number of units to make change for \( n 

**Outputs:** number of half dollars, quarter, dimes, nickels, and pennies to use \( (c_{50}, c_{25}, c_{10}, c_{5}, c_{1}) \).

**Algorithm** \textsc{MakeChange}(\( n \))

\[
\begin{align*}
c_{50} &= n \div 50 \\
n &= n \mod 50 \\
c_{25} &= n \div 25 \\
n &= n \mod 25 \\
c_{10} &= n \div 10 \\
n &= n \mod 10 \\
c_{5} &= n \div 5 \\
n &= n \mod 5 \\
c_{1} &= n \\
\text{return} & \ (c_{50}, c_{25}, c_{10}, c_{5}, c_{1})
\end{align*}
\]

Because the algorithm always performs 10 calculations, its worst-case running time is \( O(1) \).

**Proof of Optimality:** Assume that the best non-greedy solution for a given instance of the problem is \( (b_{50}, b_{25}, b_{10}, b_{5}, b_{1}) \), where \( n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_{5} + b_{1} \). We show
that the greedy solution is as good as or better than the best solution. The greedy solution is \((c_{50}, c_{25}, c_{10}, c_5, c_1)\). We want to show that \(c_{50} + c_{25} + c_{10} + c_5 + c_1 \leq b_{50} + b_{25} + b_{10} + b_5 + b_1\).

Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If \(b_{50} < c_{50}\) then \(25b_{25} + 10b_{10} + 5b_5 + b_1 \geq 50\). To satisfy the given inequality these are all the possibilities.

1. if \(b_{25} \geq 2\), replace with 1 half-dollar

2. if \(b_{25} = 1\) we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins

3. if \(b_{25} = 0\) we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If \(b_{50} = c_{50}\) and \(b_{25} < c_{25}\) then \(10b_{10} + 5b_5 + b_1 \geq 25\). These are the possibilities.

1. if \(b_{10} \geq 3\), replace with 1 quarter and 1 nickel

2. if \(b_{10} = 2\) we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter

3. if \(b_{10} = 1\) we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

4. if \(b_{10} = 0\) we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if \(b_{50} = c_{50}, b_{25} = c_{25}, b_{10} = c_{10}\), and \(b_5 < c_5\).

2) We can show that the greedy algorithm doesn’t work for all possible denominations by giving a counter-example. If \(n = 12\) and \((d_1, d_2, d_3) = (1, 6, 10)\), then the greedy algorithm would return \((c_{10}, c_6, c_1) = (1, 0, 2)\). However, the optimal solution is \((c_{10}, c_6, c_1) = (0, 2, 0)\).

3) Given a list of \(k\) coin values, \((d_1, d_2, \ldots, d_k)\), and a number \(n\), we want to find the integers \((c_{d_1}, c_{d_2}, \ldots, c_{d_k})\) such that \(n = \sum_{i=1}^{k} d_ic_{d_i}\) and that \(\sum_{i=1}^{k} c_{d_i}\) is minimal.

Our subproblems consist of the optimal change set for 1 through \(n\). To keep track of the optimal solution for each subproblem we will use an array called \(sumc\) which is indexed by subproblem. (i.e. \(sumc[i]\) contains the least number of coins needed to make change for \(i\)).

\(coin[i]\) designates which coin denomination was last used when making change for \(i\) units.

\(sumc[d_1] = 1, sumc[d_2] = 1, \ldots, sumc[d_k] = 1\)

\(sumc[i] = \min_{1 \leq j \leq k} sumc[i - d_j] + 1\)

**Inputs:** denominations \((d_1, d_2, \ldots, d_k)\), units \(n\)

**Outputs:** the count of each denomination \((c_{d_1}, c_{d_2}, \ldots, c_{d_k})\).
Algorithm MakeChange($n, (d_1, d_2, \ldots, d_k)$)

for $i \leftarrow 1$ to $n$ do
    $\text{sumc}[i] \leftarrow \infty$
for $j \leftarrow 1$ to $k$ do
    $\text{sumc}[d_j] \leftarrow 1; \text{coin}[d_j] \leftarrow j$
// calculate $\text{sumc}[i]$ for $1 \leq i \leq n$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $k$ do
        $\text{temp} \leftarrow \text{sumc}[i - d_j] + 1$
        if $\text{temp} < \text{sumc}[i]$ then
            $\text{sumc}[i] \leftarrow \text{temp}; \text{coin}[i] \leftarrow j$
// determine if it is possible to make change
if $\text{sumc}[n] = 1$ then return impossible
else // generate answer
    for $j \leftarrow 1$ to $\text{sumc}[n]$ do
        $c_{d_j} = 0$ // initialization
    // traverse through coins used to make best change
    $\text{total} \leftarrow n$
    while $\text{total} > 0$ do
        $c_{\text{coin}[\text{total}]} \leftarrow c_{\text{coin}[\text{total}]} + 1$
        $\text{total} \leftarrow \text{total} - d_{\text{coin}[\text{total}]}$
    return $(c_{d_1}, c_{d_2}, \ldots, c_{d_k})$

The running time of the above algorithm is $O(nk)$. Note that this algorithm is pseudo-polynomial.